JavaScript - AMD
Best Practices Guide

Version: 2013.11.20

This document includes best practices around developing modular applications using the asynchronous module definition (AMD) specification and compliant module loaders for frontend web development. Details on each checklist item are discussed later in the document.

Checklist

AMD Development Checklist

- ❑ Do not have extraneous dependency modules in the required modules list
- ❑ Always keep dependency module list and callback arguments organized
- ❑ Only define callback argument for directly referenced modules
- ❑ Never repeat module identifiers in your define function
- ❑ Always use consistent callback argument names
- ❑ Avoid circular dependency chains

AMD Build/Runtime Checklist

- ❑ Optimize to combine AMD dependent modules
- ❑ Use a tool like Madge to help define layers
- ❑ Balance Layer sizes and number of files
- ❑ Use nested `require`'s when loading layer modules

Discussions

It is a best practices to use modular development for large applications, and AMD is a well accepted, unopinionated, Javascript library-neutral technology that is recommended for modular front-end development. Other module systems such as CommonJS are recommended for back-end JavaScript development or OSGi for backend Java development.

The following section contains additional details related to the above checklist items.

AMD Development Checklist

Do not have extraneous dependency modules in the required modules list

During development, modules are frequently added to the define/require list. Then as refactoring or other maintenance is performed on the module's content, some of the list requirements are no longer valid. Be diligent in keeping the require list accurate. Having extra unused dependencies promotes code bloat, and complicates maintenance.

Always keep dependency module list and callback arguments organized

Keeping the dependency list well organized will aid in overall module maintenance and reduce developer confusion.

Having a mismatch between the dependency list and the callback arguments is the primary cause of AMD related defects!

The specifics on how you organize the modules is somewhat stylistic in nature. Here is a sample recommendation on organizing modules in the following order:

1. Core modules (eg. Base classes, templates, NLS) unique to this module.
2. Then alphabetically, grouping modules by prefix (ie dojo, dijit, dojox, myApp)
3. Modules not directly referenced in callback function should be listed last, with a descriptive group comment (See next topic). For example:

```
// The following modules are used by the template,
// and not directly referenced in this module
```

For large dependency lists, add a blank line every so often to break up the list. Likewise, in the callback list, match the breaks with a new line of arguments.

Only define callback argument for directly referenced modules

Do not add modules to the callback argument list that are not directly referenced within that module's definition. If a module is required, but is not directly referenced within the

module code, then place it at the end of the require/define dependency list. Do not give it a local variable name in the callback function's arguments. There should be an obvious 1 to 1 relationship between the dependency modules and the callback parameters.

```
define([
      "app/BaseClass",
      "dojo/_base/declare",
      "dijit/form/Button"  // Used by template, and not
                           // directly, so no callback ref
], function( BaseClass, declare) )
```

Never repeat module identifiers in your define function

Do not repeat module identifiers in your define function. This will likely cause syntax errors, especially if the same callback argument name is used as shown below. If different callback argument names are used, then confusion can occur on which variable to use. This could be confounded if one of the source modules is replaced with an alternate module. See the next practice on organization.

Incorrect:

```
define(["app/widget",..,"app/widget"],
function(Widget,...,Widget){...});
```

Correct:

```
define(["app/widget",...],
function(Widget,...){...});
```

Always use consistent callback argument names

Use standard callback argument names that match (or are very close) to the real dependency modules name. Typically, you'll want to use the base name of the dependency module as its callback argument name. If the dependency name has a hyphen, then convert it to camelCase for the callback argument.

Optimize to combine AMD dependent modules

Properly optimize all JavaScript and CSS. This is the single best practice for high performing apps, especially when using Dojo. Also ensure that any external libraries are pre-optimized and not in source form.

Avoid circular dependency chains

When developing custom modules, care should be taken to ensure that no circular dependencies are created between modules. For example, module "A" has a dependency on module "B". But, module "B" has its own dependency back on module

"A". Some loaders can deal with this condition, but it's best to avoid this situation entirely. This is typically the result of poor encapsulation design. When it's unavoidable to have two module rely on each other, the best solution is to create a third private base module that the top level modules can then each reference.

See the discussion on Madge below, as it will identify any circular module dependencies in your application.

AMD Build/Runtime Checklist

Optimize to combine AMD dependent modules

This is the single best recommendation for any modern web app. You must optimize (build/minify) all JavaScript modules. The details of performing the optimization -- also known as minification -- are outside the scope of this document. It is also library dependent, so read up on optimization and include it into your build flow.

For developers, they should typically always list the discrete modules in the dependency lists. The exception is for app (or subsystem) initialization. This is where you want to specifically load the generated layer files.

Defining layer files requires analysis of the app or subsystem to determine what modules and the full chain of dependency modules are actually used. Doing this by hand is unrealistic and fragile. A recommended approach would be to use a tool such as Madge, that will provide a full dependency chain for any module. See the next topic on Madge for details.

Once you know what modules define a layer, create an empty layer module that lists the desired dependencies. This layer module can typically return an empty object as shown below as layers are not generally used in callbacks. The exception to this is when creating a reusable library, in which case, you'll generally want to return the core functional module that defines the library's API implementations. Here is a sample layer definition. Notice how as a convention, it's located and defined within a "layer" directory to keep it separate from the actual code base. This prevents novice developers from trying to require it directly in other modules.

```
//-- layers/myAppLayer.js
define([
    "app/config",
    "app/init",
    "app/settings",
    "common/settings",
    "other/utilities",
```

```
    . . .
], function() {
    return {};
});
```

Again, for reusable libraries specifically, you would want this layer file to be located within the same namespace, or as a parent level top module. For example, my “Farce” library would have a top level layer module called “farce/main.js” that depends on all the other modules under the farce/ directory. It returns an object that exposes all the public API functions useable by this library. The built layer module is saved simply as “farce.js”. The end users of the library would simple load “farce.js” and have full access to my awesome library.

Once the layers have been defined, they should be loaded at an application level only. By this we mean that the layer(s) should only be referenced during app startup, or some app level navigation manager, that’s responsible for managed loading of major app subsystems. Layer modules should never be directly referenced as a dependency by any business level modules within the app.

You should not bundle discrete modules into the deployable hybrid app file that have been optimized into a production layer. They are not used and add to package bloat.

Use a tool like Madge to help define layers

There is a great open source node project called "madge" that performs a static code analysis on your source code to determine all of the dependencies. You can point it to your code, and it will show all AMD module dependencies for the entire app, or for specific modules. These results, along with a little data swizzling, can be used to create new custom layer in the build process. For example, assume you are using Dojo and want to create a custom dojo.js main layer. Using the following command line shows the most frequently used modules. You can then take the most heavily used modules and add them to the main dojo layer in the build profile.

```
$ madge  --format amd  . | egrep "dojo?|dijit" | sort -b | uniq -
c | sort -n
```

Returns a list like this (just last several items shown):

```
   8   dojox/mvc/getStateful
   9   dojo/_base/event
   9   dojo/date/stamp
   9   dojo/dom-style
  12   dijit/layout/ContentPane
  16   dojo/promise/all
```

```
  20   dojo/touch
  30   dojo/on
  32   dojo/dom-class
  38   dojo/Deferred
  41   dojo/dom-construct
  77   dojo/_base/array
  87   dojo/_base/lang
 164   dojo/_base/declare
```

This command scans the entire "app" tree -- where let's say all of your custom code resides -- and outputs all "dojo/dijit/dojox" modules required. The list is then sorted by the number of times the module is required.

Madge can do much, much more than this, so please check it out.

Balance Layer sizes and number of files

For large projects, multiple layers should be defined. Use a few base layers that contain frequent and common modules. AMD is good at loading multiple modules in parallel. General guidance is that any given layer should be <400K in size, and no more than 4 layers loaded at a time.

Use nested require's when loading layer modules

Often, during build time, multiple modules are combined into layers. This provides performance benefits, but can also break expected application startup flow. Novice developers will often attempt to bootstrap the application by issuing a require on both the layer modules as well as discrete modules contained within the layer. This results in 404 errors saying the discrete module is not found.

For example, here is a failing app startup block, where a production layer has been defined, yet the developer

```
require(["layers/myAppLayer", "app/init", "app/settings"],
function( layer, init, settings ) {
    // We never get here as the init and settings modules
    // have been removed after build and no longer exist!
    init( settings );
    . . .
});
```

The solution is to use nested require's so that we have an ordered loading of the layer module(s), and once loaded, we can then perform another require to define the local variables we need to start the app. Here is the proper startup flow for this scenario:

```
require(["layers/myAppLayer"],
function() {
    require(["app/init", "app/settings"],
    function(init, settings) {
        // All is good as the layer is loaded, and the
        // needed modules are now defined.
        init( settings );
        . . .
    });
});
```

An alternate approach is to use the synchronous require syntax. This is common in NodeJS applications, but works in Dojo and most other browser based loaders. The only caveat is that if the expected modules are not already loaded then an Error is thrown, but that's the case anyway, so it's more of a stylistic choice. The following example shows using the direct module access technique.

```
require(["layers/myAppLayer"],
function() {
    var init     = require("app/init"),
        settings = require("app/settings");

    init( settings );
    . . .
});
```